词法分析器 - 实验报告

1. 实验过程

1.1. 声明

头文件引入和输出要求的行列标号定义。代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int i = 0;
int num_lines = 1; //行
int num_cols = 1;  //列
```

1.2. 辅助定义

- 特殊字符（换行、tab、空格）的定义。

```
NEW_LINE "\n"
TAB "\t"
SPACE " "
```

- 字母和数字字符（正数、十进制、十六进制、八进制）的定义。

```
LETTER [A-Za-z]
digit [0-9]
POS_digit [1-9]

OCT_digit [0-7]
OCT_pre 0
HEX_digit [0-9a-fA-F]
HEX_pre 0x|0X
```

- 数据常量的定义（由各种进制下的数字字符组成）。

```
INTEGER (({POS_digit}{digit}*)|0)
OCT_INTEGER ({OCT_pre}{OCT_digit}*)
HEX_INTEGER ({HEX_pre}{HEX_digit}+)

CONST_NUM ({INTEGER}|{HEX_INTEGER}|{OCT_INTEGER})
```

- 标识符的定义：由字母开头的数字或字母或_组成的字符串。

```
IDENTIFIER (({LETTER}|_)({digit}|{LETTER}|_)*)
```

- 关键字的定义：c语言中各种关键字的枚举。

```
KEYWORD
main|return|if|else|for|while|do|switch|case|break|continue|const|static|sizeof|
int|unsigned|signed|static|void|auto|struct|union|inline
```

- 算符的定义：c语言中各种算符的枚举。

```
OPERATOR ("+"|"-"|"*"|"/"|"%"|"++"|"--"|"="|"=="|"!="|"<="|">="|"<"|">"|"+="|"-
="|"*="|"/="|"&"|"|"|"!"|"&&"|"||"|"~"|"^"|">>"|"<<"|"?"|":"|"."|"->")
```

- 界符的定义：c语言中各种分隔符的枚举。

```
DELIMITER (";"|","|"("|")"|"["|"]"|"{"|"}"|"#")
```

- 单行注释的定义：`//`及其后跟随的单行字符。

```
Annotation_Single_Line (("//")(.*))
```

- **多行注释的定义（难点）**：
  - 以`/*`作为开头；
  - 以`*/`结尾，结尾前可以有若干(≥0)个`*`；
  - 中间的匹配模式：由于正则表达式是贪婪匹配，会识别最远的匹配结构，这会导致在有多处多行注释时会直接匹配到最后一处注释，中间的部分会被认为所有都是注释部分。因此，需要保证匹配结构中间不能有额外的`*/`。可能的匹配模式如下：
    - 非`*`的字符；
    - 若干(≥ 1)个`*` + 但不是`/`也不是`*`作为结尾；
    - 以上两者方式的随机组合。
  - 不难发现，以上匹配模式中间不可能出现`*/`
  - 代码如下：

```
Annotation_Paragraph ("/*"(([^\*]|\*+[^\/\*])*)\*+"/")
```

1.3. 识别规则

- 按照单词符号出现顺序，依次输出：原始单词符号、种类、出现在源程序的位置（行数和列数）。
- 对于换行符、空格、tab键，单独处理。

```
{NEW_LINE} {
    ++num_lines;
    num_cols = 1;
}
{TAB} {num_cols += 4;}
{SPACE} { num_cols++;}
```

- 关键字、标识符、常量、算符、界符的处理。

```
{KEYWORD} {
    printf("%s: K, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
```

```
}
{IDENTIFIER} {
    printf("%s: I, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
}
{CONST_NUM} {
    printf("%s: C, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
}
{OPERATOR} {
    printf("%s: O, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
}
{DELIMITER} {
    printf("%s: D, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
}
```

- 注释和其他内容处理（T类）：

```
{Annotation_Single_Line} {
    printf("%s: T, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols += yyleng;
}
{Annotation_Paragraph} {
    printf("%s: T, (%d, %d)\n", yytext, num_lines, num_cols);
    while(i < yyleng)
    {
        if(yytext[i] == '\n') //换行符
        {
            num_lines++;
            num_cols = 1;
        }
        else if(yytext[i] == '\t') //tab
            num_cols += 4;
        else //其他注释语句
            num_cols++;
        i++;
    }
}
. {
    printf("%s: T, (%d, %d)\n", yytext, num_lines, num_cols);
    num_cols++;
}
```

2. 测试

编写脚本进行批量化测试：

```
flex 1.l
gcc lex.yy.c -o a
./a < ./test/0.sy > ./result/0.out
./a < ./test/1.sy > ./result/1.out
./a < ./test/2.sy > ./result/2.out
./a < ./test/3.sy > ./result/3.out
./a < ./test/4.sy > ./result/4.out
./a < ./test/5.sy > ./result/5.out
./a < ./test/6.sy > ./result/6.out
```

测试结果无误（由于篇幅较大，故不在此展示）。

3. 收获与体会

- 通过此次实验，笔者对正则表达式匹配的编写规则、匹配模式有了更深入的理解和实践经验。
- 对c语言的单词种类、语法规则更熟悉。